Sent by Hansen
FAK 3 Feb 70

ARGONNE NATIONAL LABORATORY
Argonne, Illinois 60439

Applied Mathematics Division

GRAPHIC EDITING OF STRUCTURED TEXT*

Wilfred J. Hansen

January 15, 1970

Technical Memorandum No. 190

It has been submitted for presentation at the Computer Graphics 70 Conference, Brunel University, Oxbridge, England, April 14-16, 1970.

*Work performed under the auspices of the U.S. Atomic Energy Commission.

TABLE OF CONTENTS

LIST OF FIGURES

GRAPHIC EDITING OF STRUCTURED TEXT

Wilfred J. Hansen

ABSTRACT

Emily is an interactive graphics system for creating and manipulating program texts in terms of a string representation of the tree structure that the syntax of the programming language imposes on the text. Tree nodes that require subnodes before the program is complete are displayed as non-terminal symbols. For each non-terminal symbol in turn, Emily displays a list of syntactically correct replacement strings (possibly containing both characters and lower level non-terminal symbols). The user selects the replacement needed to create the program he has in mind.

Because Emily retains the tree structure, either structural units of the text can be copied, moved, or deleted. Any unit can be represented on the display by a single identifier, so the user can view his text at any level of complexity. As a hybrid of string and tree representations, this technique provides a new way to look at and think about programs.

Several extensions to Emily are planned: interactive cross-references, dynamic syntax-controlled display formatting, and the ability to name such items as syntaxes, pieces of text, and display statuses. An important goal of the Emily project is measurement of the behavior of users. Such measurement can help describe interactive computer use, as well as assist in optimizing Emily's convenience to the user.

I. INTRODUCTION

Interactive computer systems promise to be effective tools because they can provide even the least organized human with a patient and meticulous scribe. Sitting at a console, a user ought to be able to examine a body of information from alternate viewpoints and to modify that information as required. With Sketchpad (Sutherland, 1963) graphic images can be rotated, replicated, replaced, revised, and reflected upon by merely pushing buttons and pointing a light pen. Emily is an attempt to extend such facilities to program texts. It should be possible to manipulate and modify text almost as easily as willing the action. The user should not even need a paper copy of the file. Not only might a copy be lengthy and out-of-date, but there is an orientation problem when referring back and forth between paper and display. Facilities must be available within the editing system for displaying quickly any portion of the file the user wishes to see.

Emily is an interactive syntax-controlled system for creating and modifying program texts using an IBM 2250 Graphic Display Unit. Unlike other text editors, the text is not stored and manipulated as a linear string of characters. Instead, operations are performed in terms of the tree structure imposed on the program by the syntax of the programming language. In particular, a program is created by selecting appropriate syntactic rules from those rules that apply at any particular point of the program. Since all punctuation is contained in these rules, the programmer need not be concerned with the details of punctuation. Moreover, when an identifier is required, Emily displays existing identifiers. The programmer can enter a new identifier or simply point at an old one; thus he does not have to worry about spelling.

The growth of interactive systems has spawned many text editors, but few that do much more than maintain a file in a manner analogous to shuffling cards. Editors designed for use with slow output devices such as teletype-writers are limited because they cannot present very much information in the time the user is willing to spend waiting. This means that they can only display a small portion of the context of a statement. Most early slow-device systems were designed around line numbered files. Perhaps the most advanced such editor is the Wylbur system at Stanford University (Stanford, 1968). More sophisticated slow-device editors eliminate line numbers and permit editing by context. An advanced example of such an editor is the QED editor written K. Thompson (Thompson, 1968). One of the features of this editor is that the object of a string search may be specfied to have arbitrarily complex structure.

Graphic interaction opens many possibilities for text editors. Large blocks of text can be displayed so that the context of any statement can be made clear. One of the simplest and yet most effective editors is TVEDIT by B. L. Tolliver (described in McCarthy, 1967). Each command is a single character (a special button is held down to indicate command mode); thus the user can conceive operations he wants on the screen and have them occur almost without conscious physical effort. Two graphic editors provide complex predefined structures within which the user can organize his text. The Hyper-text system by Nelson and van Dam (Carmody, 1968) permits the user to both annotate his text and provide links to other portions of the text. The text appears to the user as a network of interconnected blocks. The Stanford Research Institute's Center for the Augmentation of Human Intellect has developed an editor that allows the text to be organized into sections, sub-sections and sub-...sub-sections (Engelbart, 1968). The edited text thus has a hierarchical appearance and the editor indents the display of each subsidiary level.

Several text editors have been built that check syntax. All have one or more of these disadvantages: 1) only local line-by-line syntax is checked; 2) the editor handles only one language; 3) a typing error means listening to an error message and then, often, retyping the whole line; 4) the syntax checker occupies a large segment of core without doing much more than one run of the compiler would do. Many of these problems have been solved in a JOVIAL editor at Systems Development Corporation (Bratman, 1968). The display control in this system is table driven so any of several inexpen-sive text display devices can be used. An editor that guarantees syntac-tically correct statements is the DIALOG system (Cameron, 1967). In this primitive one-language graphic editor, the user is constrained to create correct programs because at any stage only a character that is legal can be selected as the next character of the text. As in Emily, the user of DIALOG builds his text by choosing options, although in DIALOG his choice is limited to an alphanumeric character set.

There are two reasons for implementing Emily instead of using one of these other editors. The first is to illustrate the advantages of a text editor that 'understands' the structure of what it is editing. Such an editor can, for example, ensure correct syntax while the program is being entered. Emily was originally designed as a string editor with a general purpose parser. But design of such a parser for incomplete text appeared to be a difficult problem in itself, whereas we were interested in building a usable tool in a reasonable time. Consequently, the current approach was decided upon as a method of letting the user create the structure at the same time as he creates the program. Because she understands the structure of the text, Emily can display that structure either by formatting the text, drawing diagrams or representing substructures with a single identifier. She can permit modification and reorganization by structural units. And she can facilitate file access either down the hierarchy, or via an interactive cross-reference feature.

The second reason for implementing Emily is to investigate the technique of creating programs by repeated choice from a set of alternatives. The process of writing programs as a sequential string of characters is so ingrained that it is difficult to even conceive of any other technique. A working system must be built and must be tried by people before it will be possible to evaluate Emily's approach. Other human engineering questions that must be answered include: Assuming Emily is useful, for what kind of programmer is she best suited--neophyte, expert, language designer? What training period is required in order to use Emily? What features contribute to user dissatisfaction? Which are most used? How should syntaxes be designed for Emily? How efficient is Emily compared to traditional paper-pencil-keypunch-shuffle techniques? Here efficiency is defined in terms of cost, time, and usage of scarce resources.

The initial version of Emily reported here is implemented in PL/I for an IBM 2250 Graphic Display Unit attached to an IBM 360/75. The 2250 displays characters and vectors on a 12" x 12" screen. A light pen can be used to point at an image on the screen and there is a program function keyboard with thirty-two buttons, each of which generates a unique interrupt. Characters can be typed in, under program control, from an alphanumeric keyboard; the latter generates an interrupt only when the END key is depressed. Two syntaxes have been hand coded for Emily's syntax tables: a subset of PL/I, and a language for input to a syntax table generator. Future syntaxes will be designed using Emily and implemented by processing them with this table generator.

The remainder of this paper has three parts. The first describes the capabilities of the present Emily system, while the second outlines the features that are planned for addition to Emily. The final section discusses the human engineering problems and how we plan to approach them.

II. THE CURRENT EMILY SYSTEM

For any given language, only some strings of characters constitute programs. This set of legal strings can usually be defined by a *syntax*. The characters that can appear in a program are called the *terminal symbols*. For the purpose of defining the legal strings, the syntax also introduces a set of *non-terminal symbols* that do not appear in a legal program. Finally, the syntax includes a set of *syntactic rules*, each of which specifies a string (of terminals or non-terminals) that can replace a particular non-terminal. A syntactic rule has a left-hand-side specifying a non-terminal and a right-hand-side specifying a replacement string.

As an example, Fig. 1 shows a possible syntax for part of the PL/I declaration structure. Notice that non-terminal symbols are surrounded by '<' and '>'. Usually the name inside these brackets has some relation to the semantic meaning of the string generated by the symbol. The simplest declare statement that can be generated by this syntax is 'DECLARE A FIXED BINARY;'. More complex would be 'DECLARE A FIXED BINARY, B FLOAT BINARY STATIC INTERNAL;'. The steps in the generation of the latter string are shown in Fig. 2.

In addition to defining a set of legal strings, a syntax imposes a *tree structure* on any such string. The tree for the string in Fig. 2 is shown in Fig. 3. Each syntactic rule used in the generation of the string is represented as a *node* (a rectangle) in the tree, with one pointer to a subnode for each non-terminal in the node. Emily uses this tree representation to store the file. Rather than place an entire rule in each node, a code number in the node indicates which rule generated that node. Notice that terminal symbols do not appear explicitly in this representation; they are implicit in the code number for the rule.

To an Emily user, the 2250 screen appears divided into three parts: the text, the menu, and the message area. The text occupies the upper two-thirds of the screen and shows a portion of the file being edited. The lower third of the screen is the menu and displays up to twenty options. The message area is the bottom line of the screen and is used to request operands and to display status and error messages.

Because the file may not be a complete program, the text area may include non-terminal symbols.* These are underlined to make them stand out. One of the non-terminals is designated the *current non-terminal* and is shown surrounded by a rectangle. The menu normally displays all strings that can be substituted for the current non-terminal. These strings are simply the right-hand-sides of the syntax rules that have the current non-terminal on the left.

The most usual mode of operation is *build mode*, signified by BUILD MODE in the message area. This mode is used for constructing and viewing the text. Pointing the light pen at an item in the menu substitutes that item for the current non-terminal. Pointing at a non-terminal moves the rectangle to that non-terminal so that it becomes the current non-terminal. (The menu

*When it is displayed, a non-terminal is the end (or terminal) of a branch of the tree. It is called a non-terminal because it must be replaced with a string of terminals before the program is complete.

```
Terminals:  DECLARE, ';', ',', FIXED BINARY, FLOAT BINARY,
            STATIC, AUTOMATIC, INTERNAL, EXTERNAL

Non-Terminals:  <DECL>, <IDDECL*>, <IDDECL>, <ATR>,
            <STORCL>, <SCOPE>, <ID>

Rules:
            <DECL>     ::= DECLARE <IDDECL*>;

            <IDDECL*>  ::= <IDDECL>, <IDDECL*>

                       ::= <IDDECL>

            <IDDECL>   ::= <ID> <ATR>

                       ::= <ID> <ATR> <STORCL> <SCOPE>

            <ATR>      ::= FIXED BINARY

                       ::= FLOAT BINARY

            <STORCL>   ::= STATIC

                       ::= AUTOMATIC

            <SCOPE>    ::= INTERNAL

                       ::= EXTERNAL
```

Fig. 1. Sample Syntax for <DECL>

Omission of the left-hand-side indicates that it is the same as in the preceding rule. <ID> is a special non-terminal that is replaced with any identifier name.

DECLARE $\boxed{\text{<IDDECL*>}}$;

DECLARE $\boxed{\text{<IDDECL>}}$, $\underline{\text{<IDDECL*>}}$;

DECLARE $\underline{\text{<ID>}}$ $\boxed{\text{<ATR>}}$, $\underline{\text{<IDDECL*>}}$;

DECLARE $\boxed{\text{<ID>}}$ FIXED BINARY, $\underline{\text{<IDDECL*>}}$;

DECLARE A FIXED BINARY, $\boxed{\text{<IDDECL*>}}$;

DECLARE A FIXED BINARY,
    $\boxed{\text{<ID>}}$ $\underline{\text{<ATR>}}$ $\underline{\text{<STORCL>}}$ $\underline{\text{<SCOPE>}}$;

DECLARE A FIXED BINARY,
    B $\underline{\text{<ATR>}}$ $\boxed{\text{<STORCL>}}$ $\underline{\text{<SCOPE>}}$;

DECLARE A FIXED BINARY,
    B $\boxed{\text{<ATR>}}$ STATIC $\underline{\text{<SCOPE>}}$;

DECLARE A FIXED BINARY,
    B FLOAT BINARY STATIC $\boxed{\text{<SCOPE>}}$;

DECLARE A FIXED BINARY,
    B FLOAT BINARY STATIC INTERNAL;

Fig. 2. Steps in the Generation of
DECLARE A FIXED BINARY, B FLOAT BINARY STATIC INTERNAL;

In each step the non-terminal in the rectangle is replaced to generate the next step. The order is not strictly left-to-right, but it could have been.

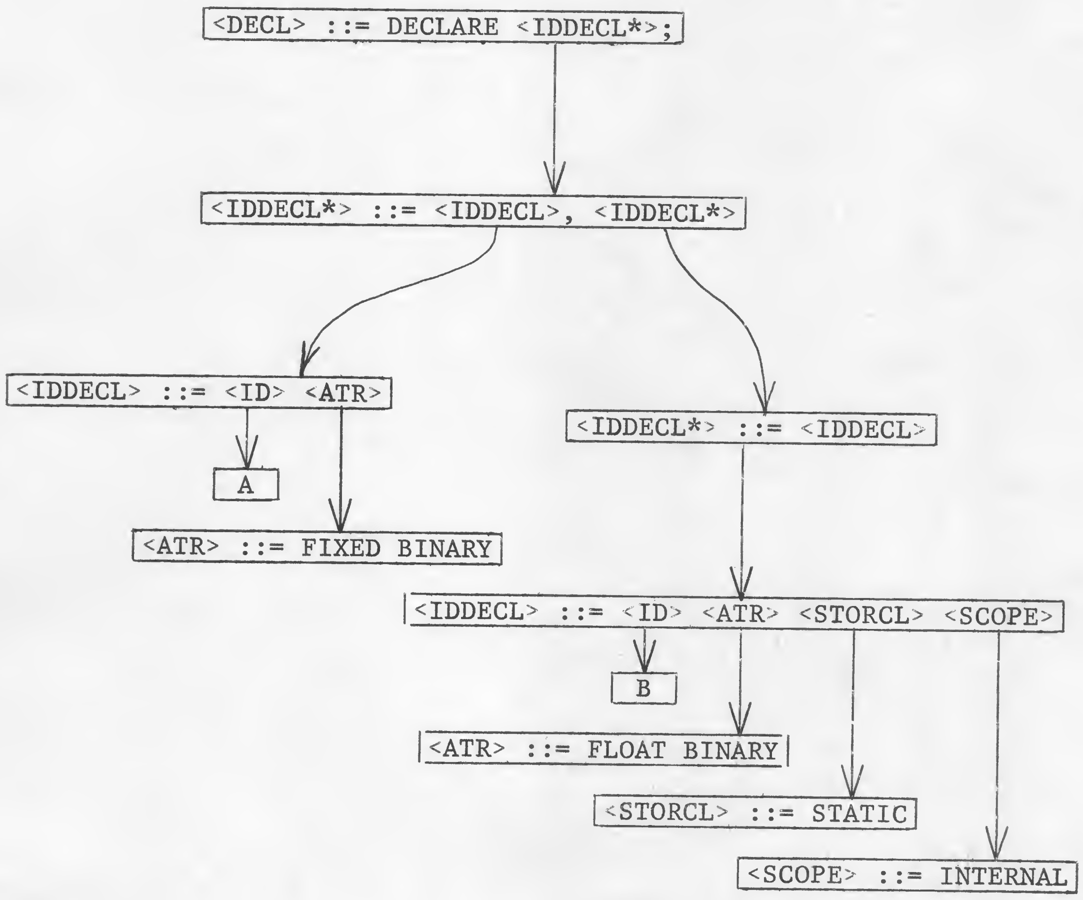

Fig. 3. Tree Structure Imposed by Syntax on the string of Fig. 2

changes accordingly.) When the current non-terminal is an identifier, the menu displays all previously entered identifiers in the required class (some of the classes for PL/I are <ID>, <LABEL>, and <ENTRYNM>). The user may select one of these, or he may enter a new identifier from the keyboard. Constants and comments are also entered from the keyboard.

To facilitate creation and modification of programs, Emily recognizes lists in the syntactic description of a language. A non-terminal representing a list has a name whose last character is '*'; for instance, <STMT*> represents a list of <STMT>'s. If the current non-terminal is a list, Emily displays all options for an element of that list and also two special options. If the user selects one of the options for the element, that option is inserted in both the file and the display. The non-terminal for the list is displayed immediately after the inserted option, so the user can continue the list. Thus, if the current non-terminal is <STMT*> and the user selects the option <SIMPLE STMT>, the display would appear as <SIMPLE STMT><STMT*>. The new current non-terminal is the first non-terminal of the option chosen, in this case, <SIMPLE STMT>.

The first of the two special options is EMPTY. Selecting this option for <STMT*> ends the list and removes <STMT*> from the display. If the syntax requires that the list have at least one element, EMPTY is not displayed as an option until the list has at least one element. The second special option is the list element itself, e.g., <STMT>. If this option is chosen <STMT> is inserted in the display before <STMT*>, but the current non-terminal remains <STMT*>. With this option, the user can create the list <STMT><STMT><STMT>... before he fills in the details of any individual <STMT>.

The flexibility of Emily's display of the file is greatly enhanced by abbreviating parts of the display with *holophrasts*. Each node in the tree generates some consecutive string in the display representation of the file. The holophrast for this string is simply the non-terminal symbol on the left-hand-side of the syntax rule that generated the string. Holophrasts are not underlined; thus they can be distinguished from non-terminal symbols in the display. For example, the eighth line in Fig. 2 might be displayed with one holophrast as 'DECLARE <IDDECL>, B $\boxed{\text{<ATR>}}$ STATIC $\underline{\text{<SCOPE>}}$;'. We emphasize that the introduction of a holophrast does not in any way modify the file, it merely modifies the display.

Holophrasts are introduced into the display by switching to CONTRACT MODE and pointing at a character in the display. That character is generated by some node of the tree. The display of the entire string generated by that node is replaced by a holophrast. For example, the comma in Fig. 3 is generated by the node corresponding to <IDDECL*> ::= <IDDECL>, <IDDECL*>. Pointing to that comma would reduce the declaration to 'DECLARE <IDDECL*>;', where <IDDECL*> is a holophrast representing 'A FIXED BINARY, B FLOAT BINARY STATIC INTERNAL'. Pointing at a holophrast in the contract mode causes the father of the indicated node to be represented by a holophrast. This new holophrast subsumes the earlier one. Pointing at a holophrast in build mode causes the node to be expanded with each of its immediate subnodes represented by holophrasts. By judicious use of contraction, the user can view his program from higher levels. He can see it as a sequence of statements or a sequence of procedures. On a much lower level, contraction makes it possible to find the principle operation of an involved arithmetic expression, or to count the number of arguments of a subroutine.

In addition to switching between the two modes described above, the program function keyboard buttons activate special functions. Most function buttons require operands in the form of light pen points or entries from the keyboard. If the user changes his mind and wishes to perform some other function or mode, he can press another button instead of responding with the required operand. Emily is very forgiving; she is designed so that most actions can easily be undone. The following provide rudimentary, but complete, facilities for displaying and modifying a file.

GO DOWN - Instead of displaying the entire program from the beginning, Emily can display any node of the tree and only its subnodes. The display node is selected by pointing the light pen at a terminal symbol generated by the node.

GO UP - The display can be moved up to any node that is a direct ancestor of the current display node. After pushing the GO UP button, the user must type in the number of levels of tree he wishes to ascend.

DELETE - A holophrast can be converted to a non-terminal by pressing DELETE and pointing at the holophrast with the light pen. Similarly, identifiers can be converted into non-terminals. The user is protected from deleting too much because he must contract the node he wishes to delete before he can delete it.

USE DUMP - The subtree deleted by DELETE is saved in a dump. When USE DUMP is pressed and the current non-terminal has the same name as the non-terminal that generated the node in the dump, the dump replaces the current non-terminal. The dump contains only the last deleted subnode and is empty after a successful USE DUMP.

INSERT ELEMENT - Elements can be inserted in--and deleted from--lists. After pressing this button the user light pens the character immediately before the location at which the element is to be inserted. If more than one list element ends at the indicated character, Emily asks the user to resolve the ambiguity by indicating (from the menu) the name of the non-terminal that generates the wanted list.

DELETE ELEMENT - An element can be deleted from a list if it is a non-terminal, a holophrast, or an identifier. (Any other element can be contracted to a holophrast). The user presses this button and points at the unwanted element. The deleted element replaces the contents of the dump and can be recalled with USE DUMP.

TERMINATE - This button saves the file on the disk and terminates Emily's execution.

Originally, we considered Emily to be an editor for the tree structure of a program. Indeed, the file is physically stored as a tree structure of nodes interconnected by pointers to subnodes. But we chose to display the file in terms of the string generated by the tree; thus Emily evolved into hybrid of tree and string editing. This hybrid appears to be a very flexible vehicle for viewing files. Strings are no longer of static length. Rather, they expand and contract as subtrees are converted between holophrasts and strings.

Achieving even the modest system described here required substantial development of data structures and interactive facilities. An Emily file is list structured; each node of the tree is represented on the disk as a block of words that contains the syntax rule number and pointers to the subnodes (one for each non-terminal on the right-hand-side of the rule). Each such pointer occupies one word and can be one of the following:

DISKNODE - pointer to another node on the disk. This pointer includes a record number and an offset to the node.

IDNODE - offset to an identifier in the identifier area. The latter is kept in core during execution.

NTNODE - contains the non-terminal number for a non-terminal that has not been replaced.

The subfields of the pointer word are accessed in PL/I by overlaying a based variable on the word. The part of the file displayed is represented in core by a set of nodes similar to those on the disk; the nodes in core contain additional information such as a pointer to the corresponding node on the disk and a pointer to the father of the node (in core). Since these nodes are maintained in core, the file has to be accessed only for changes to the display.

The heart of the interactive program structure is an attention control block containing the attention type, a pointer to an indicated node, and the number of a subnode of that node. The light pen location is decoded by table lookup in an array of x-coordinates starting at the first entry for the line identified by the y-coordinate. The node and subnode number in the attention control block are set from parallel arrays and indicate the location of the light pen selection in the data structure. The attention type is set to indicate the source of the attention. Light pen attentions decode into one of seven types depending on how the detected character was generated. Each button decodes into a separate attention type. In addition, there is the attention type NOATTN. When a routine processes an attention, it sets attention type to NOATTN. If the attention interpreter finds that the attention type is not NOATTN, it simply returns the existing attention. Thus if a routine does not wish to handle a given interrupt, it simply executes any operations required to terminate its own work and returns to the calling routine. The calling routine (usually the central dispatcher) can simply ask for another attention, and it will receive the attention that has not yet been handled.

III. FUTURE PLANS FOR EMILY

Many changes must be made to Emily before she achieves her potential as a dynamic and flexible tool for the creation of programs. These changes fall into three categories: 1) what the user sees, 2) how he can change what he sees, and 3) how he can change the file.

The display is formated by Emily. The file contains no format control characters such as end-of-line or space. Instead, the display is based on the structure of the text. Certain syntax rules include codes for how much to indent (or outdent) subsequent lines and codes for starting new lines. Thus one rule for the DO statement is

```
<STMT>  ::=  'DO;'  +6  NL  <STMT*>  +0  NL  'END;'
```

where NL is the code for a new line and +n indicates that the left margin for subsequent lines will be indented n spaces from the current left margin. Thus a DO statement would look like

```
DO;
    <STMT*>
END;
```

This scheme has several deficiencies. First of all, the entire DO statement might fit on a single line and would look awkward spread over several lines. Moreover, it is difficult to put static format codes into arithmetic syntax rules, because the same rule must apply to both long and short expressions. A third problem is that natural groupings like /*<COMMENT>*/ and (A) may be broken onto two lines because the output routines put tokens on the next line when the first line is full. The current formatter might be called a static formatter since it depends entirely on the syntax and does not take into account the actual text. A dynamic display formatter bases its decisions on both the structure of the text and the space remaining in the output area. A dynamic display formatter for LISP has been written (Hansen, 1969), and it may be possible to generalize the techniques developed in that effort.

The user may also be interested in seeing a graphic representation of the structure of the text. One form of such a display is shown in Fig. 3; the boxes represent nodes and the lines their interconnections. However, we are considering an alternative in which the entire string generated by some node is surrounded by a large box which is sub-divided to indicate the strings generated by the immediate subnodes of the node. Commands would then be available to move the big box from node to node.

'What the user can do' is vitally influenced by what he can refer to. That is, in order to operate on an object, he must have some means of naming it or pointing at it. For this reason, Emily will include a comprehensive *name facility*. A name is simply an identifier and an associated class. The following will all be nameable with different classes of names:

files
syntaxes
pieces of text
display statuses.

Names will be manipulated by the same subroutines as used for identifiers. When a name in a given class is required as an operand, Emily will display all existing names in that class. The user may either select one of these, or enter a new name from the keyboard. For example, the button to create a structure in another syntax will display the names of all syntaxes.

The present program must be expanded to give the user greater control over what he sees on the screen. The GO UP and GO DOWN buttons are fairly primitive techniques for controlling which portion of the text is displayed. At the least, GO UP must be modified so that instead of asking how many levels to ascend, it displays the non-terminal names of all predecessors of the current node and asks to which of these it should ascend. The level of the display is controlled by converting between strings and holophrasts. It may be possible to contract a string to a holophrast simply by pointing to the two ends of the string. Reversing the process, a hit on a holophrast in BUILD mode should expand it as much as possible without going off the display. This can be done in a breadth first manner, so the structure of the program is displayed before accessing the details.

The above facilities enable the user to modify his view of the file, but they do not allow him to randomly access completely different parts of the file. He must ascend to a higher level node and then descend along some other path. The name facility will permit the user to name a display status and to return to a given display at any time. Perhaps an even more important mechanism for accessing the text will be the *interactive cross-reference facility*. For any identifier, the user will be able to look at its declaration, or to look at each of its appearances, one at a time. This facility is simulated in a normal string editor by the string search capability. The user types in any string and asks the system to find all instances of the string. While this feature is more general than that proposed here, it does not distinguish between declarations, instances, comments and accidental juxtapositions of characters. Nor can string search ever hope to distinguish between identifiers with the same name, but declared in different blocks. This will be possible with Emily because she understands the structure of the text. The interactive cross-reference facility and named display statuses will provide rapid and flexible access to random parts of the text.

When he has found a portion of the file he wishes to modify, the user must have equally flexible commands to modify the file. The present deletion technique will be extended to allow a deleted structure to be placed in a named cell. Each cell will contain a subtree, the name of the non-terminal that generated the subtree, and the name of the syntax. Provision will be made for retrieving named cells from other files. Also, it will be possible for a file to include subtrees from named cells created in a syntax other than the syntax for the file. Thus, for example, a user might wish to include a lengthy comment having a syntax of his own design. As this implies, the user will be able to create syntaxes and then use them to create text. This facility will also be valuable when we are designing syntaxes for others to use.

We have also considered a nearly opposite approach to file creation. In this approach, the user would have at hand a fragment of text and the menu would display all syntax rules of which that fragment could be a part. For example, an expression would have the alternatives of being a subexpression, a subscript, an argument, or the right side of an assignment statement. Such a build-from-the-bottom approach can be implemented in Emily without undue difficulty. A further extension would be the ability to specify that any named subtree is to replace the current non-terminal. In this case, Emily might have to search through the syntax rules to find the smallest chain of rules such that the subtree can be generated by the current non-terminal.

The ultimate extension of Emily will be to provide a text processing language so the user can write small programs to modify his file. Such a language should not be difficult to implement, because the data is already structured. The language has only to provide names for the operators that manipulate Emily's internal structure. Thus, there would be a syntax for the text processing language, and the user would create named structures or named files containing statements in this language. Such a language might be similar to COGENT (Reynolds, 1965), a language which processes tree structures by using generalized analytic and synthetic assignment statements.

With the features described above, a user should be able to view his information from many perspectives and to modify as little or as much as he desires. Most of these features are illustrations of the first reason for implementing Emily. That is, they are possible because Emily 'understands' the structure of the information being edited. This understanding makes it possible to emphasize the structure as the text is displayed, to move around in the structure at will, and to modify the structure in terms of its components.

IV. MEASURING EMILY'S VALUE

The second reason for implementing Emily is to determine the value of the proposed techniques for editing programs. Most of the questions posed in the introduction are concerned with qualitative rather than quantitative judgements. Nevertheless, they are important questions and some attempt must be made to answer them.

In order to answer such human engineering questions, Emily will gather comprehensive statistics. The primary source of this information will be the actions of the user; every button-push, light-pen hit, and keyboard entry will be recorded in a special system file. This file and the initial state of the text file will be sufficient to reconstruct the entire editing session. A mode will be possible in which Emily reads this interaction file and continuously modifies the display while the user (or debugger) watches. This will be valuable for debugging and also when the user must recover a state that existed in the middle of the session.

From this file of interactions, we expect to learn many valuable facts concerning the utility of Emily's features. The frequency of use of the different functions will aid in optimizing the most used parts of the program. The ratio of file modification to file examination should yield basic facts about what programmers do when they program. For comparison, the author has kept some statistics on his own activities with pencil-paper-keypunch during the creation of this initial version of Emily. Since the author will probably be the most extensive user of Emily, these statistics may provide interesting comparisons. In addition to recording all interactions, we intend to provide an outlet for the learner to voice his frustrations with Emily. There will be a file into which he can type messages. Also, an 'I am confused' button will elict a sympathetic message from Emily as dictated by a random number generator. This will serve as a toy and will thus distinguish those seriously using Emily from those merely playing. Perhaps, we will find that a certain amount of play is necessary while learning to use a system.

An important question we hope to answer is "What is the training period necessary to learn to 'think Emily'?" That is, how long need a programmer work with Emily before he is sufficiently familiar that

a) he selects syntax rules from the menu without paying too much attention to the text area;

b) he presses buttons without looking, and without translating from a desired action to the name of a button to perform that action.

A programmer who has attained this proficiency with Emily can be said to be using it more as a tool than as a toy. There does not seem to be any good signal that a programmer has reached this level, but we hope to be able to measure proficiency in terms of reaction time and the number of mistakes and backups. Especially, we intend to look for plateaus in the learning curve, since the end of a plateau signals a transition in the way a user perceives his activities.

To some extent, the slope of learning curves is a measure of the 'naturalness' of Emily. Clearly, this does not mean that Emily is like any particular natural thing that the user has ever done. What naturalness means in Emily (and, in fact, in any interactive system) is that there should be a very few rules prescribing the behavior of either the program or the user. There must be a single point of view from which all the actions of the program appear natural. If the program is not designed with consistent behavior requirements, the user will take that much longer to figure out what is going on. (But, if he persists, he can learn any system. And once learned, that system will seem more natural than another, even though the other might require far less work. This accounts for much of the inertial persistance of non-optimal systems and languages.)

An important factor that contributes to Emily's 'naturalness' is the design of the syntax. A given language can usually be described by many different syntaxes. Simply a change in the order of the rules may help the user to find the rule he wants. More complex reorganization can change the shape of the tree without changing the string generated. For example, if the rules below replaced the rules for <IDDECL> in Fig. 1, the syntax would still generate the same strings.

```
<IDDECL> ::= <IDBASE> BINARY
         ::= <IDBASE> BINARY <STORCL> <SCOPE>
<IDBASE> ::= <ID> FIXED
         ::= <ID> FLOAT
```

Emily is designed to permit change of the syntax in a simple manner. This will permit experimentation to find out exactly what form of syntax is best.

One area in which Emily should be able to excel over traditional program creation techniques is in examination of an existing program. At present, we do not even know how a programmer does interact with his listings. The author has about twenty listings of Emily subprograms; this pile must be shuffled in order to find the relevant subprogram to answer a question. It would be much faster to ask for the subprogram by name and read it instantly. With Emily's statistics gathering machinery, we should be able to find out how often a programmer randomly accesses parts of his program. Perhaps with this information, we can get some idea of how much time is saved by using Emily.

There is, indeed, some question as whether Emily will benefit more the neophyte programmer or the programmer who is well versed in the sophisticated techniques Emily makes available. The former will be saved from remembering all the technical details of syntax, which can be extensive for a language like PL/I. But before he can use Emily, the neophyte must have a sufficient grasp of the language to understand the semantics of the syntactic alternatives. Moreover, because there are often more possibilities than can be displayed at once, the user has to know how to get to the desired option. Thus the neophyte must undergo a substantial training period simply to learn his way around the syntax, without worrying about sophisticated techniques for finding parts of his program. The design goal of Emily is a program that offers the most advantages to the practiced advanced programmer. We will assume that the user understands the syntax he is working with and has practiced the manual operations necessary to use Emily effectively.

An unexpected possibility with Emily is its potential utility in the design of new languages. A language designer is mostly concerned with the semantics of his new language. But in order to write examples in the language, it must have some form of syntax. Using Emily, the designer can build a syntax in a straightforward manner. The syntax need not even be unambiguous. Further, by modification of the syntax he can modify his language and write programs in it immediately. Because Emily keeps track of the syntax, he need not refer to it constantly while he is working.

Several changes have already been made in Emily based on limited experimental use. One fact that we have learned is that the basic cycle of selecting syntax rules and modifying the display must take place very rapidly. Even times as long as three-tenths of a second are likely to provoke frustration in the programmer. After all, he knows what he wants to do and feels that he could be working more quickly with pencil and paper, even though this might not be true, since Emily can create many characters per interaction. The present program displays the next set of syntax rules before modifying the display. The programmer can select a rule before the display is completely rebuilt, and this saves some time. But the computer is so fast that the display is built just about the time the programmer decides what he wants. In this case, the text flicks on and then off again. This is very distracting, so the programmer is more likely to wait for both the rules and the text display. Since the contract mode proved to be confusing, some human engineering has already gone into its definition. The menu displays options for the current non-terminal and, if the user selects one of these, Emily automatically switches to the build mode. Emily also switches to the build mode after any deletion. This is because a frequent (and annoying) sequence of operations was: go to contract mode, contract string to be deleted, delete string, then try to change current non-terminal without having returned to build mode.

Emily looks to the day when every programmer has a console in his office. We hope that our results will be of some benefit in the design of the required systems.

V. ACKNOWLEDGEMENTS

I am grateful for the advice and encouragement of John C. Reynolds and William F. Miller.

VI. REFERENCES

(Bratman, 1968) Bratman, Harvey, Hiram G. Martin, and Ellen C. Perstein. Program composition and editing with an on-line display. AFIPS Proc. V. 33 pt. 2 (FJCC) 1968, pp. 1349-1360.

(Carmody, 1968) Carmody, Steven, Theodor H. Nelson, David Rice, and Andries Van Dam. A Hypertext Editing System for the /360. Brown University, October, 1968 (unpublished).

(Cameron, 1967) Cameron, Scott H., Duncan Ewing, and Michael Liveright. DIALOG: A conversational programming system with a graphical orientation. Comm. ACM V. 10, 6 (June, 1967), pp. 349-357.

(Engelbart, 1968) Engelbart, Douglass C., and William K. English. A research center for augmenting human intellect. AFIPS Proc. V. 33 pt. 1 (FJCC) 1968, pp. 395-410.

(Hansen, 1969) Hansen, Wilfred J. PRETTIERPRINT for LISP360. Argonne National Laboratory, Applied Mathematics Division Technical Memorandum No. 182, May, 1969 (unpublished).

(McCarthy, 1967) McCarthy, John, Dow Brian, Gary Feldman, and John Allen. THOR - a display based time sharing system. AFIPS Conf. Proc. V. 30 (SJCC) 1967, p. 623-633.

(Reynolds, 1965) Reynolds, J. C. Cogent Programming Manual. Argonne National Laboratory report no. ANL-7022. Argonne, Illinois, March, 1965.

(Stanford, 1968) Stanford University Computing Center. Wylbur Reference Manual (Appendix E of Campus Facility Users Manual). February, 1968.

(Sutherland, 1963) Sutherland, I. E. Sketchpad: A man-machine graphical communication system. T-296, Lincoln Lab., M.I.T., Lexington, Mass., 1963.

(Thompson, 1968) Thompson, K. QED Text Editor. Bell Telephone Laboratories, Murray Hill, New Jersey, March, 1968 (unpublished).